JDBC & File I/O

Advanced Java Programming

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

FileReader and FileWriter

Character-based streams for reading and writing text files:

```
// Writing to a file
try (FileWriter writer = new FileWriter("output.txt")) {
    writer.write("Hello, Zanzibar!\n");
    writer.write("Java File I/O is powerful.\n");
}  // auto-closed

// Reading from a file
try (FileReader reader = new FileReader("output.txt")) {
    int ch;
    while ((ch = reader.read()) != -1) {
        System.out.print((char) ch);
    }
}
```

Limitations

- Reads/writes one character at a time — very slow for large files
- No line-based reading capability
- Always wrap with `BufferedReader`/`BufferedWriter` for performance

BufferedReader and BufferedWriter

```
// Writing with BufferedWriter
try (BufferedWriter writer = new BufferedWriter(
        new FileWriter("students.txt"))) {
    writer.write("Ali,CS,3.8");
    writer.newLine();
    writer.write("Fatma,IT,3.5");
    writer.newLine();
    writer.write("Said,CS,2.9");
}

// Reading with BufferedReader (line-by-line)
try (BufferedReader reader = new BufferedReader(
        new FileReader("students.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        String[] parts = line.split(",");
        System.out.printf("Name: %s, Dept: %s, GPA: %s%n",
            parts[0], parts[1], parts[2]);
    }
}
```

Why Buffered?

try-with-resources

Automatically closes resources implementing `AutoCloseable`:

```
// Before Java 7 - manual close in finally block
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("data.txt"));
    String line = reader.readLine();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try { reader.close(); }
        catch (IOException e) { e.printStackTrace(); }
    }
}

// Java 7+ - try-with-resources (clean and safe)
try (BufferedReader reader = new BufferedReader(
        new FileReader("data.txt"))) {
    String line = reader.readLine();
    System.out.println(line);
} catch (IOException e) {
    e.printStackTrace();
}
```

Path and Files Classes

Java NIO.2 (since Java 7) provides a modern, powerful file API:

```
import java.nio.file.*;

// Creating Path objects
Path path1 = Path.of("data", "students.txt");
Path path2 = Paths.get("/home/user/documents/report.txt");

// Path operations
System.out.println(path2.getFileName());     // report.txt
System.out.println(path2.getParent());       // /home/user/documents
System.out.println(path2.getRoot());         // /
System.out.println(path2.toAbsolutePath()); // full absolute path
System.out.println(path1.resolve("grades.csv")); // data/students.txt/grades.csv

// File existence and type checks
System.out.println(Files.exists(path1));
System.out.println(Files.isDirectory(path1));
System.out.println(Files.isRegularFile(path1));
System.out.println(Files.size(path1));  // file size in bytes
```

Files — Reading and Writing

```
import java.nio.file.*;
import java.util.*;

// Write all lines at once
List<String> lines = List.of("Ali,CS,3.8", "Fatma,IT,3.5", "Said,CS,2.9");
Files.write(Path.of("students.csv"), lines);

// Read all lines into a List
List<String> readLines = Files.readAllLines(Path.of("students.csv"));
readLines.forEach(System.out::println);

// Read entire file as a single String
String content = Files.readString(Path.of("students.csv"));

// Read as a Stream (lazy, efficient for large files)
try (Stream<String> stream = Files.lines(Path.of("students.csv"))) {
    stream.filter(line -> line.contains("CS"))
          .map(line -> line.split(",")[0])
          .forEach(System.out::println);  // Ali, Said
}

// Write a string
Files.writeString(Path.of("note.txt"), "Hello from NIO!");
```

Files — Directory Operations and Copy/Move

```java
// Create directories
Files.createDirectory(Path.of("output"));
Files.createDirectories(Path.of("output/reports/2024"));

// Copy and move files
Files.copy(Path.of("source.txt"), Path.of("backup.txt"),
    StandardCopyOption.REPLACE_EXISTING);
Files.move(Path.of("old.txt"), Path.of("archive/old.txt"),
    StandardCopyOption.REPLACE_EXISTING);

// Delete
Files.delete(Path.of("temp.txt"));        // throws if not exists
Files.deleteIfExists(Path.of("temp.txt")); // returns boolean

// List directory contents
try (Stream<Path> paths = Files.list(Path.of("."))) {
    paths.filter(Files::isRegularFile)
         .forEach(System.out::println);
}

// Walk directory tree recursively
try (Stream<Path> walk = Files.walk(Path.of("src"))) {
    walk.filter(p -> p.toString().endsWith(".java"))
```

Object Serialization

Serialization converts an object to a byte stream for storage or transmission:

```
import java.io.*;

public class Student implements Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private String department;
    private double gpa;
    private transient String password;  // not serialized!

    public Student(String name, String department, double gpa, String password) {
        this.name = name;
        this.department = department;
        this.gpa = gpa;
        this.password = password;
    }

    @Override
    public String toString() {
        return name + " (" + department + ") GPA: " + gpa
               + " [pwd: " + password + "]";
    }
```

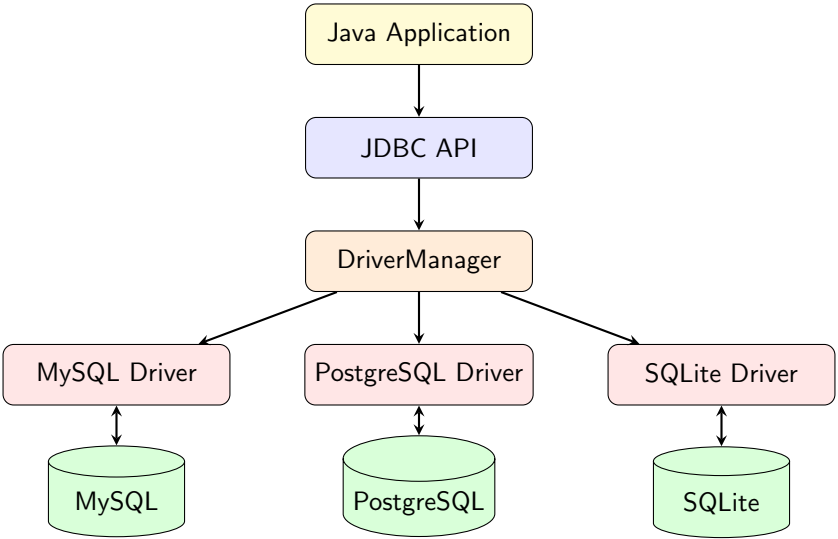
Writing and Reading Serialized Objects

```
// Serialize (write) objects to a file
try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("students.ser"))) {
    oos.writeObject(new Student("Ali", "CS", 3.8, "secret123"));
    oos.writeObject(new Student("Fatma", "IT", 3.5, "pass456"));
    System.out.println("Objects serialized successfully.");
}

// Deserialize (read) objects from a file
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("students.ser"))) {
    Student s1 = (Student) ois.readObject();
    Student s2 = (Student) ois.readObject();
    System.out.println(s1);  // Ali (CS) GPA: 3.8 [pwd: null]
    System.out.println(s2);  // Fatma (IT) GPA: 3.5 [pwd: null]
    // password is null because it was declared transient!
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

Key Points

serialVersionUID ensures version compatibility, transient fields are excluded from

JDBC Architecture



Core JDBC Interfaces

Interface/Class	Description
`DriverManager`	Manages database drivers; creates connections
`Connection`	Represents a session with the database
`Statement`	Executes static SQL queries
`PreparedStatement`	Executes parameterized SQL queries (prevents SQL injection)
`ResultSet`	Holds the result of a SELECT query; iterate with `next()`
`SQLException`	Exception thrown for database access errors

JDBC URL Format

`jdbc:<subprotocol>:<subname>`

Examples:

```
jdbc:mysql://localhost:3306/mydb
jdbc:postgresql://localhost:5432/mydb
jdbc:sqlite:students.db
```

Establishing a Connection

```
import java.sql.*;

public class JDBCConnection {
    // Connection parameters
    private static final String URL  = "jdbc:mysql://localhost:3306/university";
    private static final String USER = "root";
    private static final String PASS = "password";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }

    public static void main(String[] args) {
        try (Connection conn = getConnection()) {
            if (conn != null) {
                System.out.println("Connected to database!");
                System.out.println("Database: " +
                    conn.getMetaData().getDatabaseProductName());
                System.out.println("Version: " +
                    conn.getMetaData().getDatabaseProductVersion());
            }
        } catch (SQLException e) {
            System.err.println("Connection failed: " + e.getMessage());
```

Statement — CREATE and INSERT

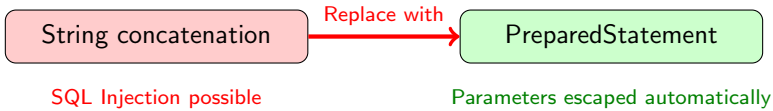
```
try (Connection conn = getConnection();
     Statement stmt = conn.createStatement()) {

    // Create table
    String createSQL = """
        CREATE TABLE IF NOT EXISTS students (
            id INT PRIMARY KEY AUTO_INCREMENT,
            name VARCHAR(100) NOT NULL,
            department VARCHAR(50),
            gpa DOUBLE
        )""";
    stmt.executeUpdate(createSQL);
    System.out.println("Table created.");

    // Insert data
    stmt.executeUpdate(
        "INSERT INTO students (name, department, gpa) VALUES ('Ali', 'CS', 3.8)");
    stmt.executeUpdate(
        "INSERT INTO students (name, department, gpa) VALUES ('Fatma', 'IT', 3.5)");
    System.out.println("Data inserted.");

} catch (SQLException e) {
    e.printStackTrace();
```

Statement — SELECT with ResultSet

```
try (Connection conn = getConnection();
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT * FROM students")) {

    System.out.printf("%-5s %-15s %-10s %-5s%n",
        "ID", "Name", "Dept", "GPA");
    System.out.println("-".repeat(40));

    while (rs.next()) {
        int id          = rs.getInt("id");
        String name     = rs.getString("name");
        String dept     = rs.getString("department");
        double gpa      = rs.getDouble("gpa");

        System.out.printf("%-5d %-15s %-10s %-5.1f%n",
            id, name, dept, gpa);
    }

} catch (SQLException e) {
    e.printStackTrace();
}
```

SQL Injection — The Danger

```
// VULNERABLE CODE - Never do this!
String userInput = "Ali'; DROP TABLE students; --";
String sql = "SELECT * FROM students WHERE name = '" + userInput + "'";
stmt.executeQuery(sql);
// Executes: SELECT * FROM students WHERE name = 'Ali'; DROP TABLE students; --'
```



Rule

Never concatenate user input into SQL strings. **Always** use `PreparedStatement` with parameter placeholders (?).

PreparedStatement — Safe CRUD Operations

```java
// INSERT with PreparedStatement
String insertSQL = "INSERT INTO students (name, department, gpa) VALUES (?, ?, ?)";
try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)) {
    pstmt.setString(1, "Amina");
    pstmt.setString(2, "CS");
    pstmt.setDouble(3, 3.9);
    int rows = pstmt.executeUpdate();
    System.out.println(rows + " row(s) inserted.");
}

// SELECT with PreparedStatement
String selectSQL = "SELECT * FROM students WHERE department = ? AND gpa >= ?";
try (PreparedStatement pstmt = conn.prepareStatement(selectSQL)) {
    pstmt.setString(1, "CS");
    pstmt.setDouble(2, 3.0);
    try (ResultSet rs = pstmt.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getString("name") + " - " + rs.getDouble("gpa"));
        }
    }
}

// UPDATE
```

Transactions — commit and rollback

```
Connection conn = getConnection();
try {
    conn.setAutoCommit(false);  // start transaction

    PreparedStatement debit = conn.prepareStatement(
        "UPDATE accounts SET balance = balance - ? WHERE id = ?");
    debit.setDouble(1, 500.0);
    debit.setInt(2, 1);
    debit.executeUpdate();

    PreparedStatement credit = conn.prepareStatement(
        "UPDATE accounts SET balance = balance + ? WHERE id = ?");
    credit.setDouble(1, 500.0);
    credit.setInt(2, 2);
    credit.executeUpdate();

    conn.commit();  // both succeed -> commit
    System.out.println("Transfer successful.");

} catch (SQLException e) {
    conn.rollback();  // something failed -> undo all changes
    System.err.println("Transfer failed, rolled back: " + e.getMessage());
} finally {
```

Batch Processing

Execute multiple SQL statements in a single round-trip to the database:

```
String sql = "INSERT INTO students (name, department, gpa) VALUES (?, ?, ?)";
try (Connection conn = getConnection();
     PreparedStatement pstmt = conn.prepareStatement(sql)) {

    conn.setAutoCommit(false);

    String[][] data = {
        {"Hassan", "IT", "3.2"}, {"Khadija", "CS", "3.7"},
        {"Omar",   "IT", "3.0"}, {"Salma",   "CS", "3.4"},
        {"Yusuf",  "IT", "2.8"}
    };

    for (String[] row : data) {
        pstmt.setString(1, row[0]);
        pstmt.setString(2, row[1]);
        pstmt.setDouble(3, Double.parseDouble(row[2]));
        pstmt.addBatch();  // add to batch
    }

    int[] results = pstmt.executeBatch();  // execute all at once
    conn.commit();
    System.out.println("Inserted " + results.length + " records.");
```

Complete JDBC DAO Pattern Example

```
public class StudentDAO {
    private Connection conn;

    public StudentDAO(Connection conn) { this.conn = conn; }

    public List<Student> findByDepartment(String dept) throws SQLException {
        String sql = "SELECT * FROM students WHERE department = ?";
        List<Student> students = new ArrayList<>();
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, dept);
            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    students.add(new Student(
                        rs.getInt("id"),
                        rs.getString("name"),
                        rs.getString("department"),
                        rs.getDouble("gpa")
                    ));
                }
            }
        }
        return students;
    }
```

Summary

File I/O

- BufferedReader/Writer for text files
- try-with-resources for safe cleanup
- NIO: Path, Files for modern file operations
- Files.lines() for stream-based reading

Serialization

- Serializable interface
- ObjectOutputStream / ObjectInputStream
- `transient` for sensitive fields
- `serialVersionUID` for versioning

JDBC

- DriverManager for connections
- PreparedStatement to prevent SQL injection
- ResultSet for query results
- Transactions: commit/rollback
- Batch processing for bulk operations

Best Practices

Always use try-with-resources for I/O and JDBC. Never concatenate user input into SQL. Use transactions for multi-step database operations. Prefer NIO for new projects.